Wrocław, 26 maja 2024

Paradygmaty Programowania Obiektowego

Laboratorium 5

Kolekcje, iteratory, przeciążanie operatorów, funkcje zaprzyjaźnione, rzutowanie

prowadzący: *mgr inż. Marta Lampasiak*

1 Wprowadzenie

Kolekcje

Kontenery (kolekcje) są dostępne poprzez tak zwaną bibliotekę kontenerów, która jest ogólnym zbiorem klas i algorytmów pozwalających na łatwą implementację typowych struktury danych. Zalicza się do nich między innymi: kolejki, listy, stosy.

Istnieją różne rodzaje klas kontenerów: sekwencyjne, asocjacyjne oraz nieuporządkowane kontenery asocjacyjne (od C++11). Najbardziej popularnymi, z którymi spotyka się każdy na początku swojej przygody z programowaniem są oczywiście kontenery sekwencyjne, w których elementy uporządkowane są jeden po drugim. Jednocześnie do elementu można odwołać się poprzez indeks lub iterator. W momencie wstawiania elementu do kontenera nie jest badana jego zawartość w celu określenia pozycji, zostaje on po prostu od razu wstawiony.

Kolekcją, z którą każdy powinien mieć do czynienia jest oczywiście `vector`, czyli dynamiczna tablica o zmiennym rozmiarze. Warto dodać, że każda kolekcja posiada operujące na niej metody: zwracające odpowiednie parametry lub wykonujące dane operacje.

Iteratory

Iterator to uogólnienie wskaźnika, który pozwala programowi C++ pracować z różnymi strukturami danych (np. kolekcjami) w jednolity sposób. Stanowi on obiekt umożliwiający dostęp do elementu przechowywanego w kolekcji.

W celu posługiwania się iteratorami w kolekcji należy dostarczyć:

1. definicję iteratora,
2. funkcję zwracającą iterator na pierwszy element `begin()`,
3. funkcję zwracającą iterator na ostatni element `end()`,
4. funkcję dereferencji, pobrania wskaźnika i inkrementacji.

Przeciążanie operatorów

Przeciążanie operatorów (ang. *operator overloading*), nazywane zamiennie również przeładowaniem, pozwala nadawać operatorom nowe znaczenie, dzięki czemu mogą być one wykorzystane w odniesieniu do obiektów zdefiniowanych klas (implementacja klasy przechowującej złożony typ danych z przeciążonymi operatorami).

Przeciążenia operatora dokonuje się definiując funkcję operatorową, której nazwa składa się:

- ze słowa `operator`,
- z występującego po słowie `operator` symbolu tego operatora, np. `+`, `-`, `*`, itd.

Jednocześnie przynajmniej jeden z parametrów musi być obiektem klasy. Musi być to obiekt, a nie wskaźnik do obiektu. Istnieje cała lista operatorów, które mogą zostać przeciążone. Natomiast są również operatory, które stanowią wyjątki, np: `.`, `.*`, itd...

Operator może być funkcją globalną lub metodą klasy. W tym drugim przypadku musi być niestatyczną metodą klasy, dla której pracuje. Jeśli operator jest funkcją globalną,to ma ona zawsze o jeden parametr jawny więcej niż ten sam operator w postaci metody.

W celu wyświetlania obiektów można przeciążyć operator `<<`. Przeciążenie operatora można zrealizować przez funkcję zwracającą referencję do strumienia oraz przyjmującą jako argument referencję do strumienia i obiekt do wyświetlenia. Używanie referencji do strumienia wynika z konieczności jego modyfikacji. Funkcja `operator<<` może być funkcją globalną – jeśli nie korzysta ze składników prywatnych klasy lub publiczną składową funkcją zaprzyjaźnioną `friend` – jeśli korzysta ze składników prywatnych klasy.

Po co w ogóle przeciążać operatory? Przede wszystkim upraszcza to notację złożonych wyrażeń.

Deklaracja przyjaźni

Funkcja, która jest przyjacielem klasy, ma dostęp do wszystkich jej prywatnych i chronionych składowych. To klasa deklaruje, które funkcje są jej przyjaciółmi. Deklaracja przyjaźni może się pojawić w dowolnej sekcji i jest poprzedzona słowem kluczowym `friend`.

Rzutowanie

static_cast Rzutowanie to jest powszechnie stosowane do konwersji typów niepolimorficznych, takich jak prymitywne typy danych oraz do rzutowania w górę w hierarchii klas (czyli np. obiektu klasy pochodnej na obiekt klasy bazowej). Wykonuje on konwersję typów w czasie kompilacji. Posiada następującą składnię:

static_cast<target-type>(expression)

dynamic_cast Rzutowanie używane ze wskaźnikami i referencjami w hierarchiach klas obejmujących polimorfizm (klasy z funkcjami wirtualnymi). Używany jest głównie do konwersji w dół (konwertowania wskaźnika lub referencji klasy bazowej na klasę pochodną). Zapewnia bezpieczeństwo typu, przeprowadzając kontrolę w czasie wykonywania w celu sprawdzenia poprawności konwersji. Jeżeli konwersja nie jest możliwa, `dynamic_cast` zwraca wskaźnik pusty `nullptr` (w przypadku konwersji wskaźników) lub zgłasza wyjątek `bad_cast` (w przypadku konwersji referencji). Posiada następującą składnię:

dynamic_cast<target-type>(expression)

2 Zadania

Utwórz klasę reprezentującą **element ekwipunku wojownika** w pewnej grze. Każdy element opisany jest trzema parametrami liczbowymi, typu *double*, które stanowią pola `prywatne` klasy i reprezentują: obronę, siłę zadawanych obrażeń, szansę na obrażenia krytyczne. Elementy ekwipunku można do siebie dodawać przez co wytwarza się element o innych, szczególnych właściwościach.

Tworząc klasę pamiętaj o implementacji konstruktora/konstruktorów, *getterów*. Tam, gdzie jest to wskazane proszę używać słowa kluczowego `const` (dla argumentów, dla funkcji) oraz dbać o to, co powinno zostać przekazane poprzez referencję. Pamiętaj również o odpowiednim podziale na pliki źródłowe i nagłówkowe. Definicje „dłuższych” funkcji zapisuj poza ciałem klasy.

Zadania na ocenę 3.0

Należy:

1. utworzyć przeciążenie operatora `<<`, jako **funkcję składową** w celu wyświetlenia obiektu powyższej klasy,
2. utworzyć przeciążenie operatora `>>`, jako **funkcję składową** w celu umożliwienia wpisywania parametrów obiektu powyższej klasy,
3. utworzyć przeciążenie operatora `+`, jako **funkcję składową** klasy umożliwiającą dodanie do siebie dwóch obiektów.

Sprawdź działanie wszystkich zaimplementowanych funkcjonalności i zaprezentuj je.

Zadania na ocenę 4.0

Kontynuuj tworzenie programu. Utwórz klasę reprezentującą zbiór obiektów wcześniejszej klasy, a więc utwórz klasę, w której znajdzie się baza danych wraz z funkcjami na niej operującymi. Klasa posiada więc pole prywatne, którym jest **kontener na dane** `vector` przechowujący obiekty. Następnie utwórz:

1. przeciążenie operatora `+=` jako funkcję składową, która umożliwia dodawanie do kontenera `vector` kolejnych obiektów (w celu dodawania ich do wektora wykorzystaj odpowiednią funkcję biblioteczną dedykowaną kolekcji `vector`).
2. przeciążenie operatora `<<` jako funkcję składową klasy reprezentującej zbiór, która umożliwia wyświetlenie poszczególnych elementów kontenera `vector`. Zastosuj **iterator** w celu przejścia po elementach.

W głównej pętli programu utwórz obiekt klasy reprezentującej zbiór elementów ekwipunku oraz zaprezentuj działanie poszczególnych funkcjonalności, a więc: dodaj dwa elementy ekwipunku do kontenera `vector` będącego częścią obiektu reprezentującego ich zbiór (bazę danych) używając przeciążenia operatora `+=`, a następnie wyświetl zawartość tego obiektu.

Zadania na ocenę 5.0

Utwórz klasę reprezentującą legendarny element ekwipunku, która dziedziczy po klasie reprezentującej ekwipunek (dziedziczenie publiczne). Dodatkowo niech posiada ona pole `siła_runiczna` typu *double* oraz utwórz przeciążenie operatora `<<`, jako **funkcję składową** w celu wyświetlenia obiektu powyższej klasy.

Następnie w głównej pętli programu: utwórz obiekty reprezentujące: element ekwipunku i legendarny element ekwipunku, a później dokonaj rzutowania statycznego pierwszego obiektu na drugi (używając `static_cast`). Zaprezentuj poprawność uzyskanych efektów.